TUBITAK-UZAY Compulsory Summer Internship Project

OPTIMIZATION ADDITIONAL REPORT

TUBITAK-UZAY-MultiCommSim

2025 SUMMER

EDİZ ARKIN KOBAK – Computer Engineer

TABLE OF CONTENTS

1. **Introduction** 2
2. **Initial Architecture** 2
   - **2.1. General Structure** 2
   - **2.2. Initial Diagram** 3
3. **Post-Optimization Architecture** 3
   - **3.1. Removing the Router** 3
   - **3.2. Flask API Changes** 4
   - **3.3. Post-Optimization Diagram** 4
4. **Port Management and TCP Multiplexing** 5
   - **4.1. Docker Network Model and Isolation** 5
   - **4.2. Bridge Network: Simnet** 5
   - **4.3. Multiple Server Containers Listening to the Same Port** 6
   - **4.4. Port Mapping (Host ↔ Container)** 6
   - **4.5. TCP Multiplexing & Socket Distribution sample scenario** 7
   - **4.6. Unlimited Peer Management with a Single IP and Single Port** 7
   - **4.7. Reasons for not using alternatives** 8
   - **4.8. Gains in General Architecture** 8
5. **Diagrams and Images** 9
   - **5.1. High Level Architecture** 9
   - **5.2. Client-Server Messaging Sequence Diagram** 10
   - **5.3. UI Images** 10
6. **Achievements** 13
7. **Conclusion** 13

1. Introduction

Bu rapor, TÜBİTAK UZAY tarafına geliştirilen MultiCommSim projesinin optimizasyon sürecini, mimari değişiklikleri, teknik kazanımları ve Docker altyapısında port paylaşımı gibi gelişmiş uygulamaları detaylı biçimde belgelemektedir.

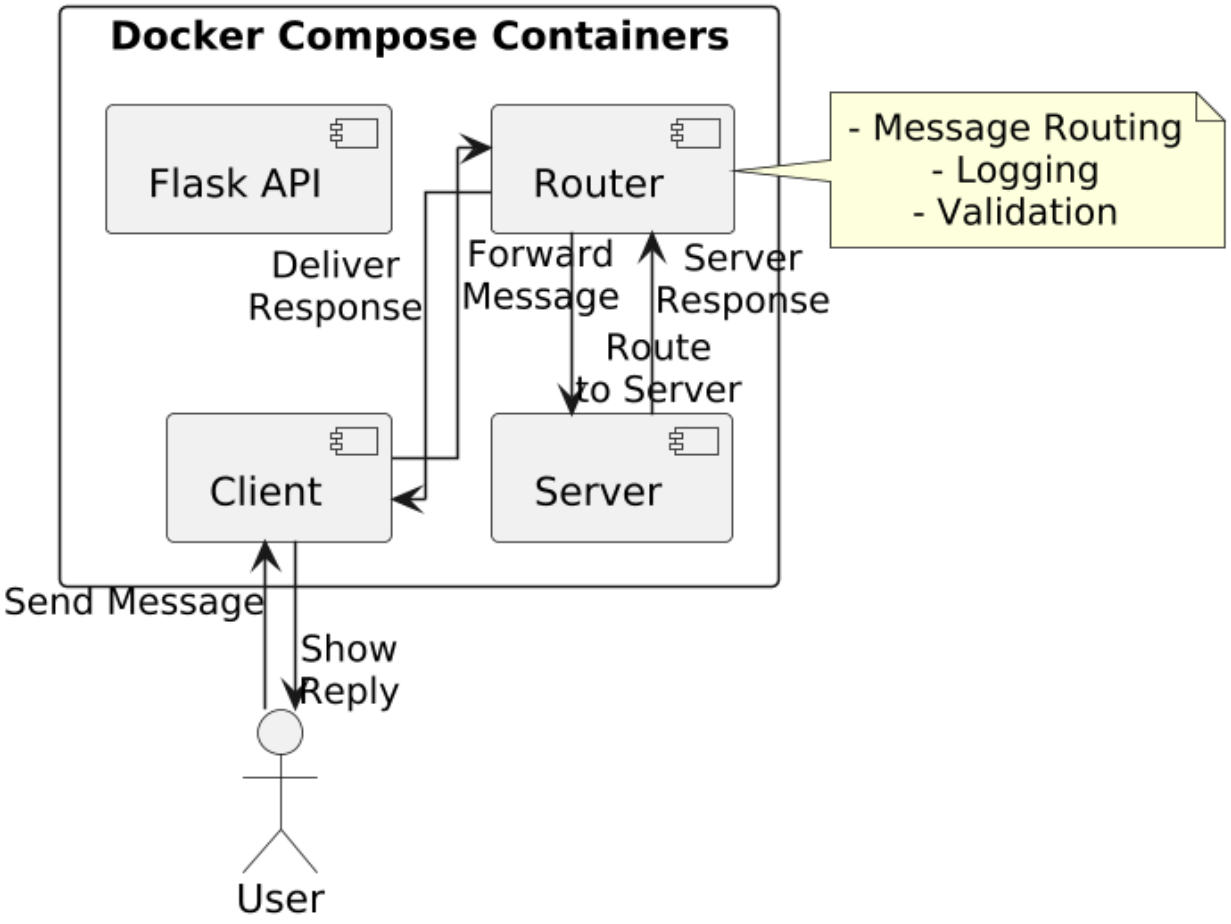
Başlangıçta router temelli mimari ile geliştirilen sistem, performans ve uygulanabilirlik sorunları nedeniyle optimize edilerek router bileşeni kaldırılmış ve client-server iletişimi doğrudan yapılabilir hâle getirilmiştir. Bu sayede sistem daha yalın, esnek ve yatayda ölçeklenebilir bir hâl almıştır.

2. Initial Architecture

2.1 General Structure

- Sistem üç ana Java bileşeninden oluşuyordu:
  - `*client/*`
  - `*server/*`
  - `*router/*`
- `*router*`, *client*'tan gelen mesajları alıp doğru `*server*` container'ına yönlendirmekle sorumluydu.
- Flask API, Docker üzerinden bu container'ları başlatıyor ve loglarını çekerek React arayüzüne sunuyordu.
- Mesaj formatı `*router/models/*Message.java` sınıfı üzerinden **JSON** ile tanımlanıyordu.
- Ortak socket tabanlı TCP bağlantı yönetimi, `*router/utils/*` içinde yer alıyordu.

2.1 Initial Diagram



---

3. Post-Optimization Architecture

3.1 Removing the Router

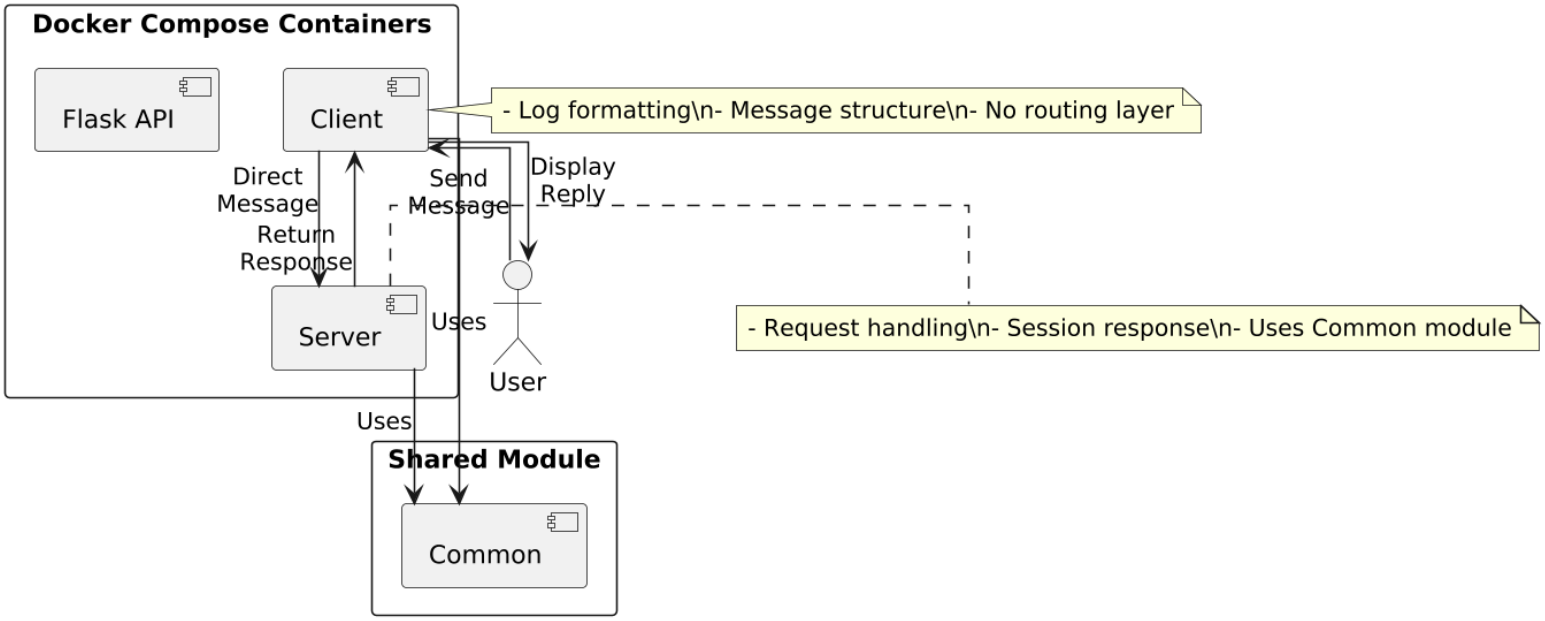
- ***router***/ bileşeni koddan ve mimariden tamamen çıkarıldı.
- Router'ın üstlendiği:
    - Mesaj yönlendirme
    - Session takibi
    - TCP socket açma

  görevleri client/ ve server/ uygulamaları arasında dağıtıldı.

3.2 Flask API Changes

- ***/create-peer*** yerine artık ***/create-server*** ve ***/create-client*** uç noktaları mevcut.
- Flask, doğrudan **java-message-server** ve **java-message-client** container'larını yaratıyor.
- Flask tarafında global bir server_counter ile benzersiz ID veriliyor (örn. **server-1, server-2…**).

3.3 Post-Optimization Diagram



4. Port Management and TCP Multiplexing

Sistemin optimize edilen versiyonunda tüm `server` container'ları **aynı port (6003)** üzerinde dinleme yapmaktadır. Normal şartlarda birden fazla işlem aynı IP ve portu dinleyemezken, Docker altyapısı bu sınırlamayı aşarak her container'a bağımsız bir çalışma alanı sunar. Bu bölümde bu durumun nasıl mümkün olduğunu detaylı biçimde açıklıyoruz.

4.1 Docker Network Model and Isolation

Docker container'ları kendi işletim sistemlerinden izole edilmiş mini sanal makineler gibi çalışır. Her container:

- Kendi **process space** (**PID namespace**)
- Kendi **dosya sistemi**
- Ve en önemlisi, kendi **ağ arayüzü** (**network namespace**) ile birlikte gelir.

Bu ağ izolasyonu sayesinde her container:

- Kendi içinde `**0.0.0.0:6003**` portunu dinleyebilir.
- Aynı host üzerinde başka bir container da aynı portu kullanabilir.
- Çakışma yaşanmaz çünkü bu portlar **container'a özeldir**.

4.2 Bridge Network: Simnet

Proje içinde kullanılan `**simnet**` adlı Docker bridge ağı:

- Tüm container'ları aynı **Layer 2** sanal ağ üzerinde çalıştırır.
- Her container'a otomatik olarak özel bir **IP (`172.x.x.x`)** verir.
- DNS çözümleme sağlar: `**server-1**`, `**client-1**` gibi container isimleri, **IP** adreslerine çözülür.

Bu yapı sayesinde:

- Client, `**server-3:6003**` gibi bir bağlantı gerçekleştirebilir.
- Docker arka planda `**server-3**` container'ının **IP** adresine yönlendirme yapar.

4.3 Multiple Server Containers Listening to the Same Port

Docker her container'ın içinde çalışan işlemleri:

- **izole bir ağ alanı** içinde çalıştırır,
- Bu nedenle `**container-A**` ve `**container-B**` aynı anda `**0.0.0.0:6003**` portunu dinleyebilir.

Yani:

server-1 içinde çalışan process

java-message-server → binds to 0.0.0.0:6003 (inside container)

server-2 içinde de aynısı yapılabilir

4.4 Port Mapping (Host ↔ Container)

Docker dış dünyadan erişim gerektiğinde port yönlendirme (port mapping) yapar:

```
ports:
  - "32866:6003"
```

Bu yapı, host'taki **localhost:32866** adresinin container içindeki **6003** portuna yönlendirilmesini sağlar.

Ama bu projede:

- Host'tan erişim gerekmediği için **port mapping yapılmaz**.
- Tüm iletişim **Docker içi DNS ve TCP** ile olur.

4.5 TCP Multiplexing & Socket Distribution sample scenario

Sistemde aynı portu dinleyen onlarca **server** container'ı olmasına rağmen, bağlantı şu mantıkla sağlanır:

Örnek Senaryo:

1. Flask API, **server-1** ve **server-2** container'larını başlatır.
2. Her ikisi de kendi içinde **6003** portunu dinler.
3. UI'den Client oluşturulup bağlantılı server'ı olarak **server-2** seçildiğinde:
    - Flask, bir **client** container başlatır.
    - Container içindeki Java uygulaması:

Java

Socket socket = new Socket(serverHost, serverPort);

→ new Socket("server-2", 6003);

Komutu ile bağlantı kurar.

- Docker, **server-2** ismini çözer ve **IP**'sine yönlendirir.
- Aynı port üzerinden doğrudan bağlantı sağlanır.

Buradaki kritik nokta: 6003 portu her container için izoledir ve Docker DNS, doğru container'a yönlendirme yapar.

4.6 Unlimited Peer Management with a Single IP and Single Port

Gerçek donanım üzerinde bu sistem çalıştırıldığında:

- Tüm container'lar tek bir fiziksel cihazda bulunur.
- Her biri Docker'ın izole ağı içinde aynı IP ve aynı portu kullanır.
- Bu yapı sayesinde **tek bir fiziksel IP + tek port (6003)** ile onlarca eş zamanlı bağlantı yapılabilir.

Avantajları:

- Herhangi bir NAT, reverse proxy veya proxy routing yapılandırmasına ihtiyaç duyulmaz.
- Bağlantı yönetimi çok daha sadeleşir.
- Dağıtık sistemlerde performans ve gecikme açısından iyileşme sağlar.

4.7 Reasons for not using alternatives

Alternatif	**Neden Tercih Edilmedi?**
Reverse Proxy (Nginx)	İç haberleşmede ekstra yapılandırma ve latency getirir.
Host Port Pooling	Yönetimi karmaşık, çakışma ihtimali yüksek.
WebSocket Communication	Stateful yapı, container restart durumlarında kırılgan yapı.
External Load Balancer	Bu proje için gereksiz karmaşıklık ve maliyet.

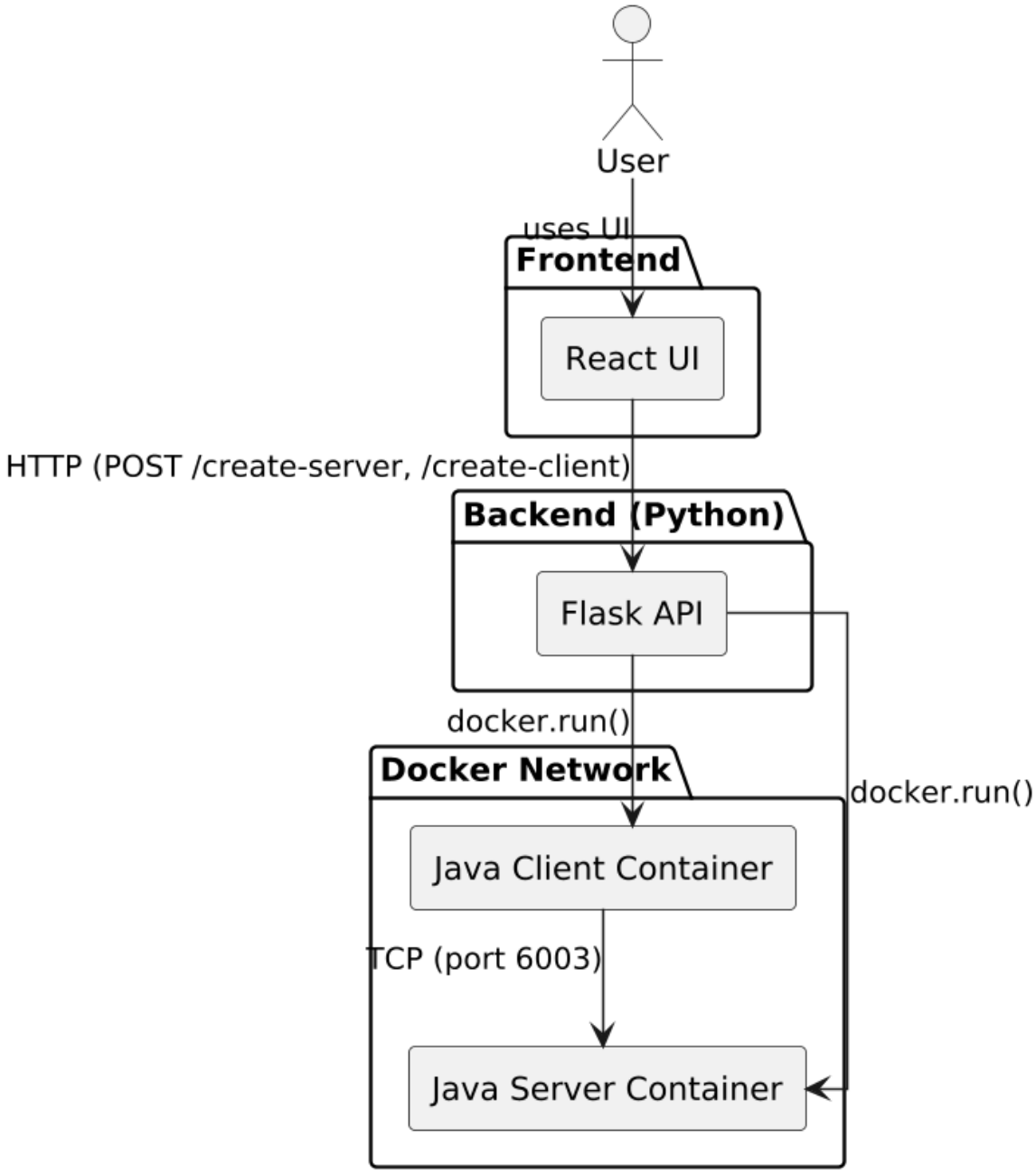
4.8 Gains in General Architecture

Docker-tabanlı bu yapı sayesinde sistem:

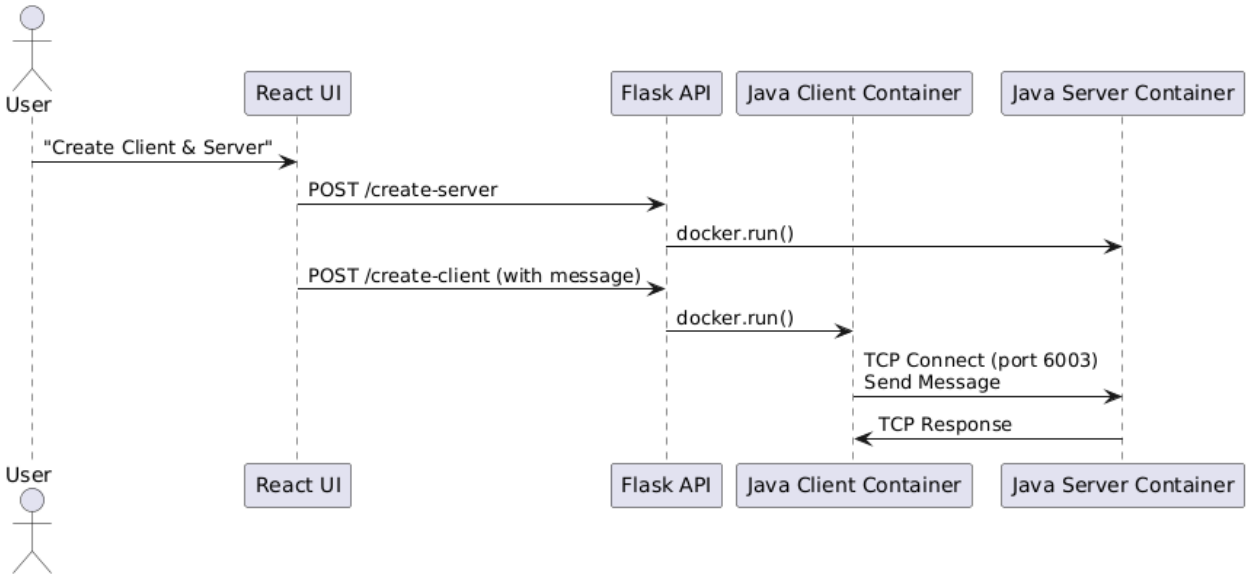
- Gerçek zamanlı olarak **çok sayıda peer'i** tek port üzerinden yönetebilir.
- Dağıtık haberleşme uygulamaları için **ölçeklenebilir**, **verimli** ve **taşınabilir** bir platform haline gelir.
- Uydu, IoT, siber güvenlik ve benzeri alanlarda bu mimari büyük avantaj sağlar.

5. Diagrams and Images

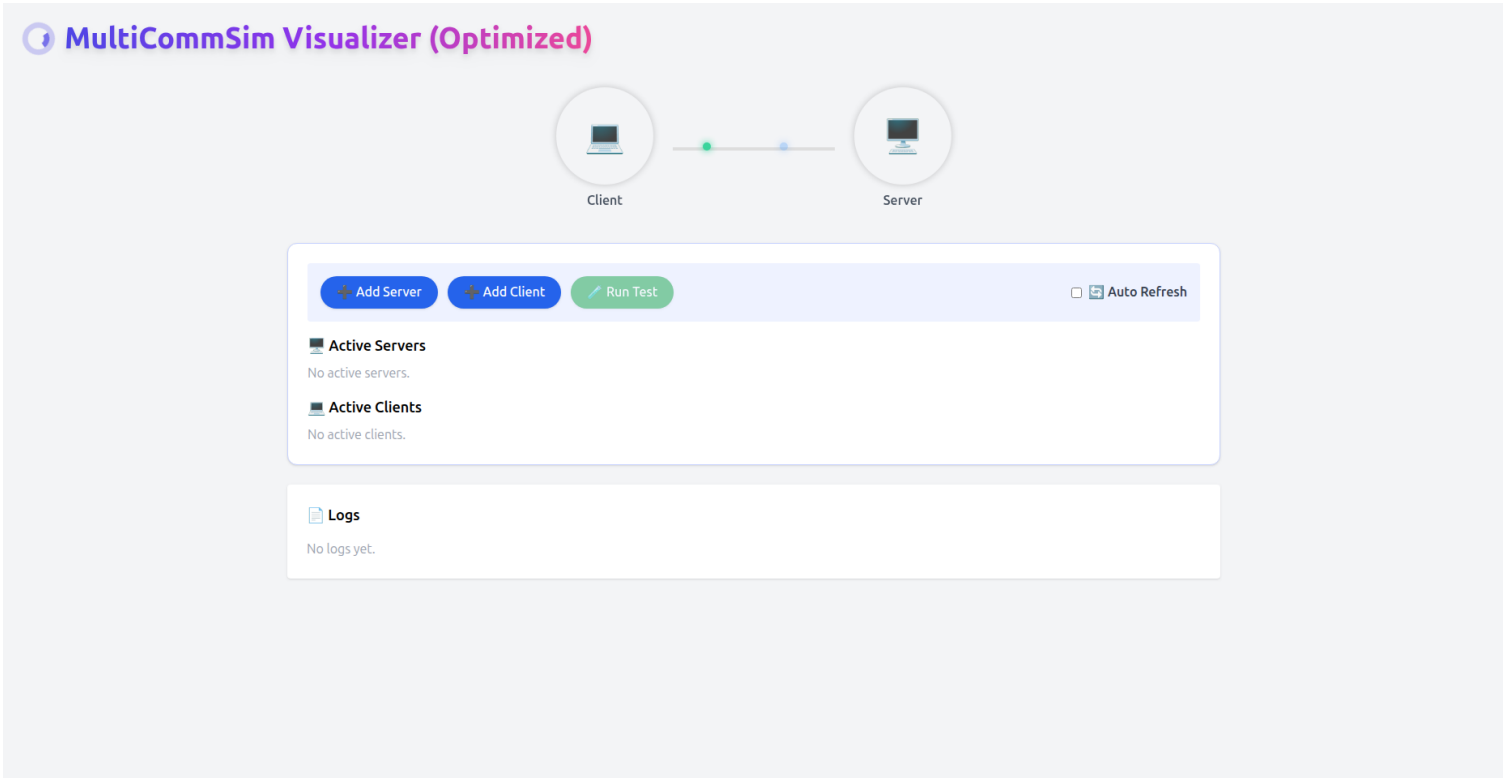
5.1 High Level Architecture

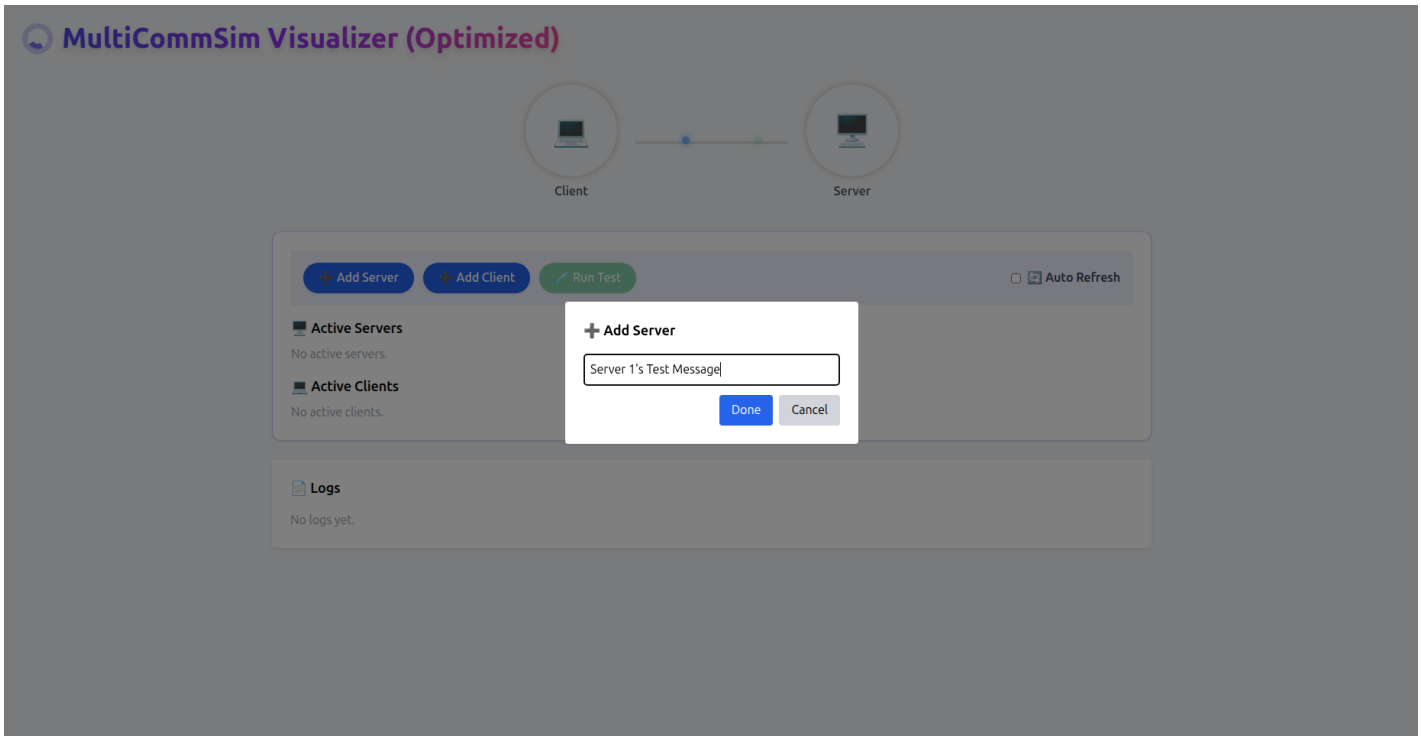


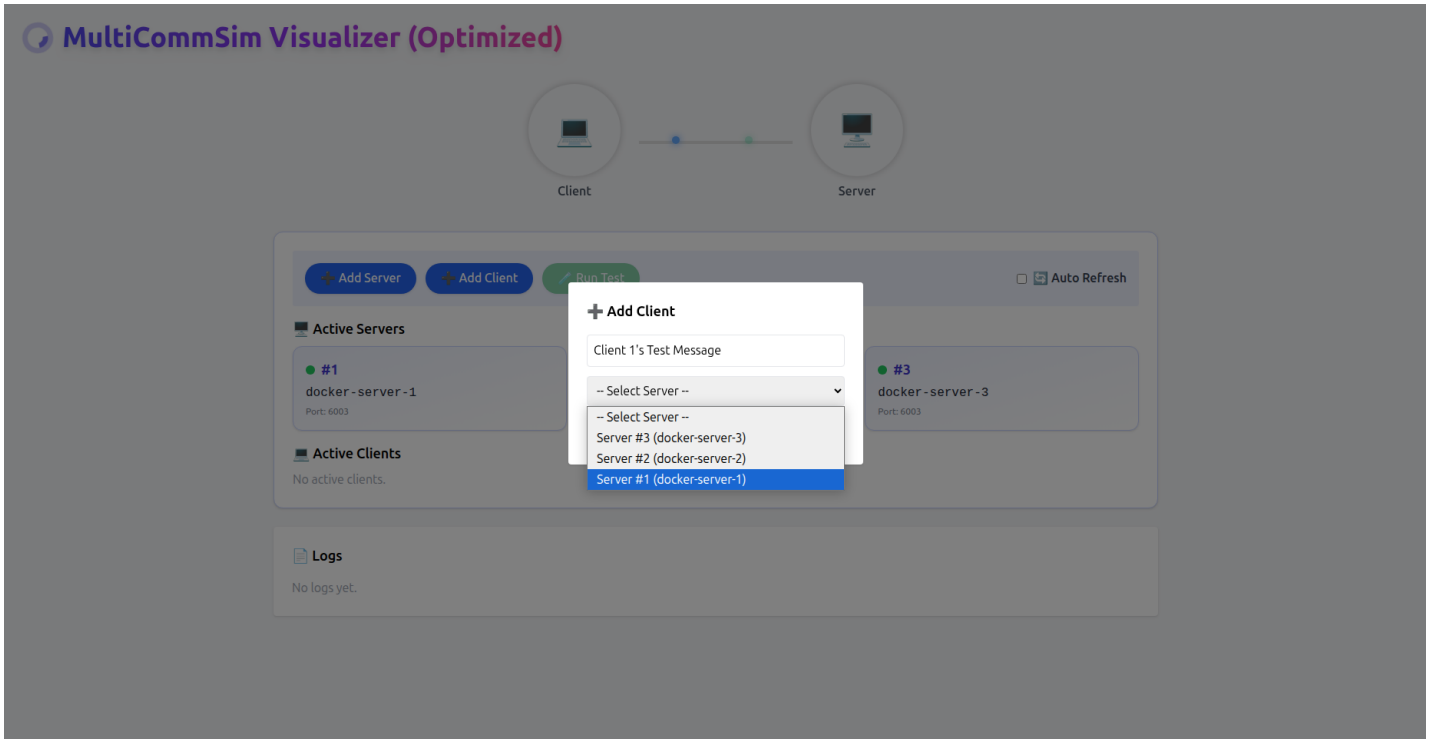
5.2 Client-Server Messaging Sequence Diagram

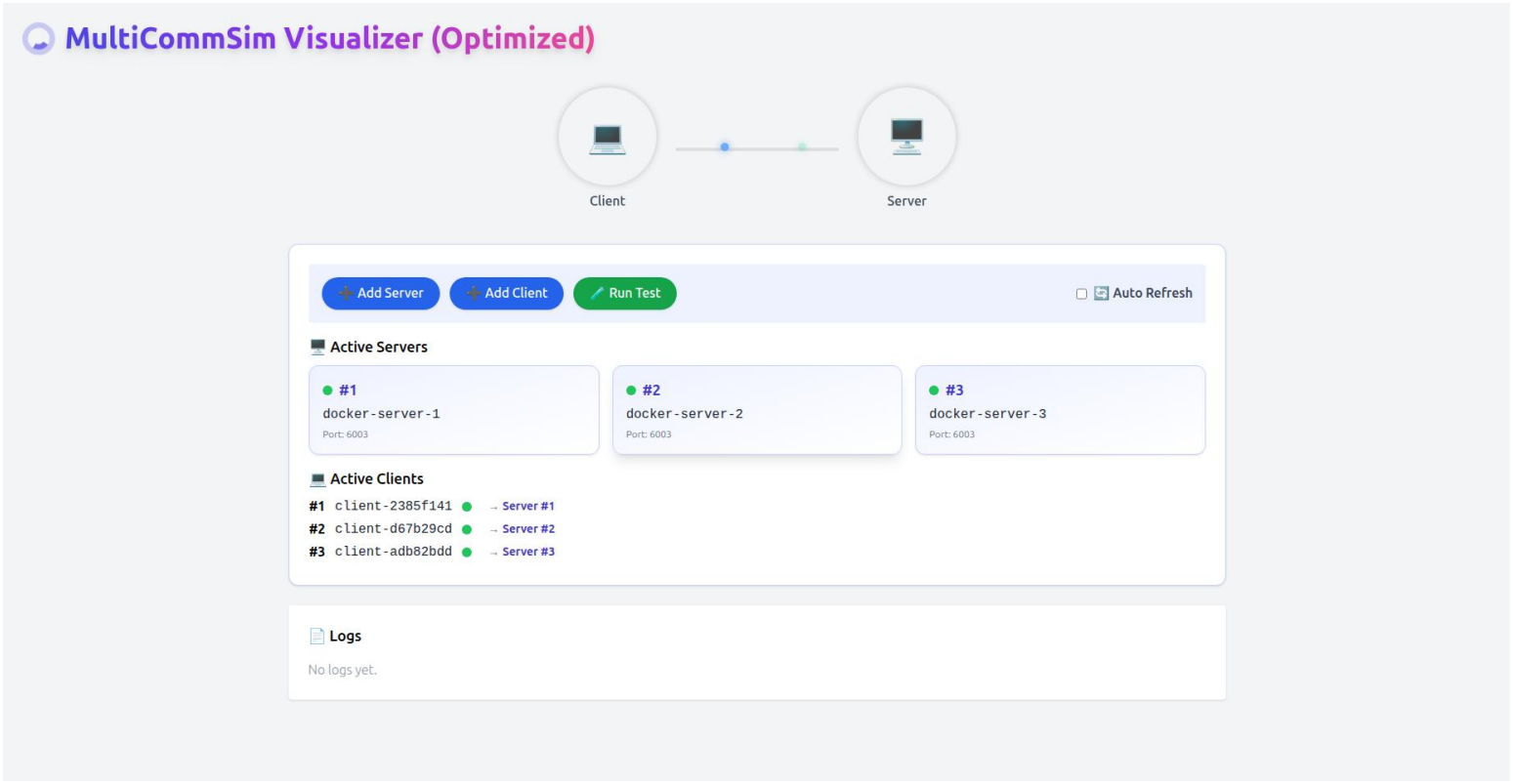


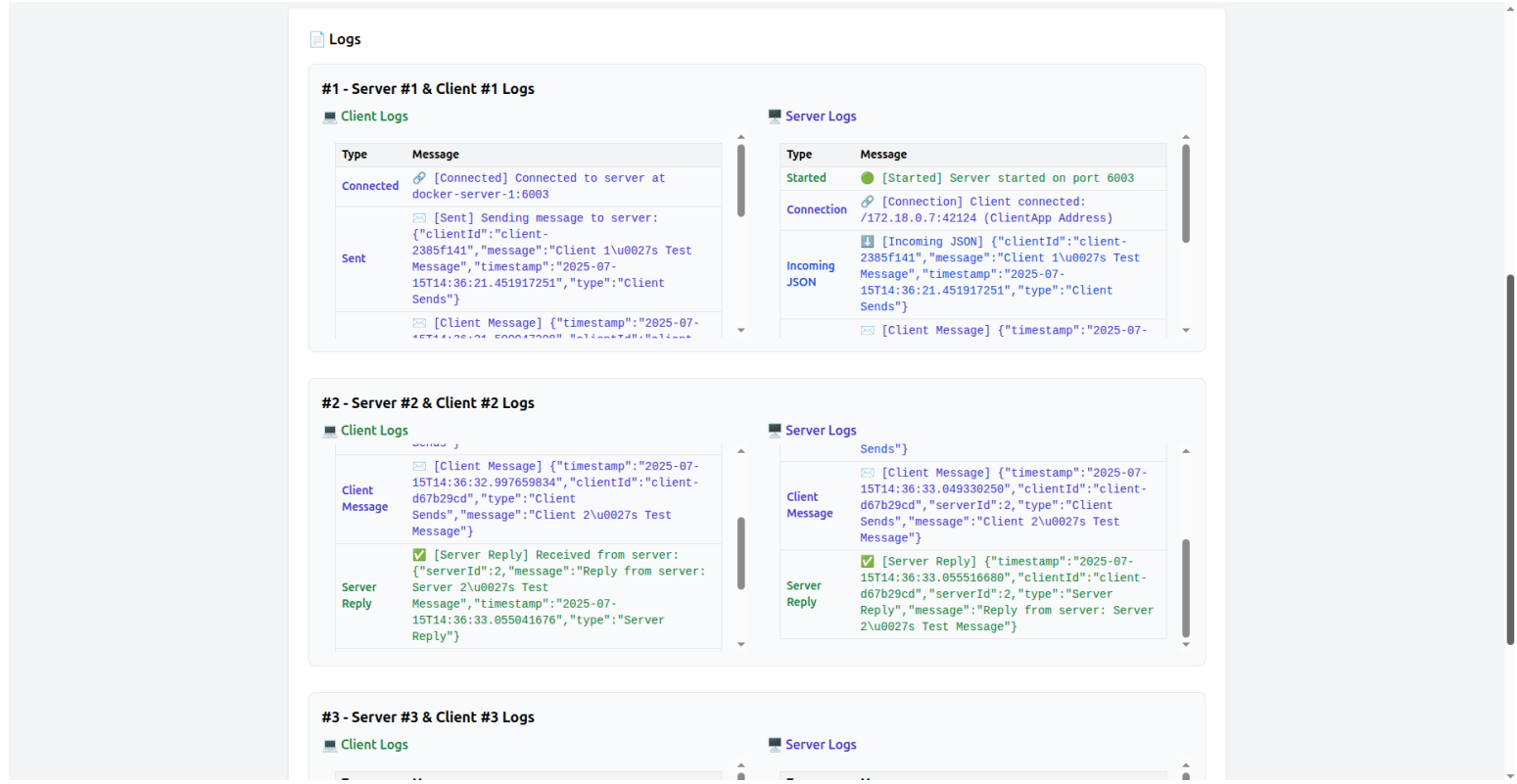
5.3 UI Images




MultiCommSim Visualizer (Optimized)
Client
Server
Add Server
Add Client
Run Test
Auto Refresh
Active Servers
No active servers.
Active Clients
No active clients.
Add Server
Server 1's Test Message
Done
Cancel
Logs
No logs yet.


MultiCommSim Visualizer (Optimized)
Client
Server
Add Server
Add Client
Run Test
Auto Refresh
Active Servers
#1
docker-server-1
Port: 6003
#3
docker-server-3
Port: 6003
Active Clients
No active clients.
Add Client
Client 1's Test Message
-- Select Server --
-- Select Server --
Server #3 (docker-server-3)
Server #2 (docker-server-2)
Server #1 (docker-server-1)
Logs
No logs yet.


MultiCommSim Visualizer (Optimized)
Client
Server
Add Server
Add Client
Run Test
Auto Refresh
Active Servers
#1
docker-server-1
Port: 6003
#2
docker-server-2
Port: 6003
#3
docker-server-3
Port: 6003
Active Clients
#1 client-2385f141 → Server #1
#2 client-d67b29cd → Server #2
#3 client-adb82bdd → Server #3
Logs
No logs yet.


Logs
#1 - Server #1 & Client #1 Logs
Client Logs
Type
Message
Connected
[Connected] Connected to server at docker-server-1:6003
Sent
[Sent] Sending message to server: {"clientId":"client-2385f141","message":"Client 1\u0027s Test Message","timestamp":"2025-07-15T14:36:21.451917251","type":"Client Sends"}
[Client Message] {"timestamp":"2025-07-
Server Logs
Type
Message
Started
[Started] Server started on port 6003
Connection
[Connection] Client connected: /172.18.0.7:42124 (ClientApp Address)
Incoming JSON
[Incoming JSON] {"clientId":"client-2385f141","message":"Client 1\u0027s Test Message","timestamp":"2025-07-15T14:36:21.451917251","type":"Client Sends"}
[Client Message] {"timestamp":"2025-07-
#2 - Server #2 & Client #2 Logs
Client Logs
Client Message
[Client Message] {"timestamp":"2025-07-15T14:36:32.997659834","clientId":"client-d67b29cd","type":"Client Sends","message":"Client 2\u0027s Test Message"}
Server Reply
[Server Reply] Received from server: {"serverId":2,"message":"Reply from server: Server 2\u0027s Test Message","timestamp":"2025-07-15T14:36:33.055041676","type":"Server Reply"}
Server Logs
Sends"}
Client Message
[Client Message] {"timestamp":"2025-07-15T14:36:33.049330250","clientId":"client-d67b29cd","serverId":2,"type":"Client Sends","message":"Client 2\u0027s Test Message"}
Server Reply
[Server Reply] {"timestamp":"2025-07-15T14:36:33.055516680","clientId":"client-d67b29cd","serverId":2,"type":"Server Reply","message":"Reply from server: Server 2\u0027s Test Message"}
#3 - Server #3 & Client #3 Logs
Client Logs
Server Logs

6. Achievements

Kategori	Eski Yapı (Router'lı)	Yeni Yapı (Optimizasyon Sonrası)
Mesaj Yönlendirme	Router	Doğrudan client → server
Socket Yönetimi	Router tarafından yönetiliyor	Server/Client kendi socket'ini yönetiyor
Loglama	Router üzerinden	Her bir container kendi logunu tutuyor
Yatay Ölçeklenebilirlik	Zayıf (tek router)	Güçlü (sınırsız peer)
Docker Karmaşıklığı	Yüksek	Sade ve modüler yapı

---

7. Conclusion

Bu optimizasyon sayesinde:

- Router'ın sistemden çıkarılmasıyla birlikte tekil hata noktası ve gecikme kaynakları ortadan kalktı.
- Docker'ın izole ağ mimarisi sayesinde, tüm container'lar 6003 portunu sorunsuz kullanabiliyor.
- Sistem daha modüler, sade ve yatayda kolay ölçeklenebilir hâle geldi.

Prepared by: Ediz Arkın Kobak

Date: 14.07.2025